

```

### getODE -- a wrapper for scan, to return the ode source file as a vector of
###      strings.
getODE <- function(s)
{
  fnm <- paste(s,".ode",sep="")
  if (!file.exists(fnm))
  {
    stop(paste("File",fnm,"does not exist"))
  }

  scan(fnm,what=character(0),strip.white=TRUE,blank.lines.skip=FALSE,
    sep="\n",quiet=TRUE)
}

### Return a vector that contains the location of each /* string
### in pode

getStarts <- function(pode, starts=0)
{
  if (starts[length(starts)]+1 < nchar(pode)) {
    nextstart <- regexpr("//",
      substr(pode,starts[length(starts)]+1,nchar(pode)))
    if (nextstart == -1) {
      if (starts[1] == 0) starts[-1] else starts
    }
    else getStarts(pode, c(starts, nextstart+starts[length(starts)]))
  }
}

### Return a vector that contains the location of each */ string
### in pode
getStops <- function(pode, stops=0)
{
  if (stops[length(stops)]+1 < nchar(pode)) {
    nextstart <- regexpr("//",
      substr(pode,stops[length(stops)]+1,nchar(pode)))
    if (nextstart == -1) {
      if (stops[1] == 0) stops[-1] else stops
    }
    else getStops(pode, c(stops, nextstart+stops[length(stops)]))
  }
}

### stripComments takes as input a vector of strings (like that output by
### getODE), and returns the input minus comments.
stripComments <- function(ode)
{
  ## First, strip out /* */ style comments. These can be multi-line
  cstart <- regexpr("//",ode)
  cstop <- regexpr("//",ode)
  ## Find the lines that are totally within slash-star comments.
  ## These will just be deleted after we deal with the lines
  ## partially in comments.
  inSlashStar <- as.logical(cumsum(cstart > -1) - cumsum(cstop > -1))
  keep <- !inSlashStar | ((cstart > -1) | (cstop > -1))
  ## Delete the slash-star comments that begin and end on the same line
}

```

```

ode <- gsub("\\\\*.*\\\\/", "", ode)
## Delete the initial partial comments
iniPart <- cstart > -1 & cstop == -1
termPart <- cstart == -1 & cstop > -1
if (any(iniPart))
  ode[iniPart] <- ifelse(cstart[iniPart] > 1,
    substr(ode[iniPart], 1,
      cstart[iniPart] - 1),
    "")
## Delete the terminal partial comments
if (any(termPart))
  ode[termPart] <- ifelse(cstop[termPart] + 2 < nchar(ode[termPart]),
    substr(ode[cstart > -1], cstop[termPart] + 2,
      nchar(ode[termPart])),
    "")
## Now delete the lines that were totally within comments
ode <- ode[keep]
## Delete the hash comments; find hash marks, and delete to end of line
hash <- regexpr("\#?",ode)
ode[hash > -1] <- substr(ode[hash > -1], 1, hash[hash > -1] - 1)
## Delete blank lines (lines with only spaces or tabs, or null)
ode <- ode[-grep("^[ \t]*$",ode)]
## Delete all leading whitespace
ode <- sub("^[ \t]*","",ode)
## Delete the trailing whitespace
ode <- sub("[ \t]*$","",ode)
ode
}

#### fixpow
####
#### walk a parse tree, replacing "^" with pow
####

fixpow <- function(x){
  if (length(x) == 1) {
    return(x)
  } else {
    if (x[[1]] == "^") x[[1]] <- quote(pow)
    for (i in 2:length(x)) x[[i]] <- fixpow(x[[i]])
    return(x)
  }
}
#### all.funcs - a function augmenting all.vars and all.names, that only returns
#### all the functions in an expression. It is defined as all names less the variables
#### and arithmetic operators

all.funcs <-function(e) {
  n1 <- all.names(e, unique=TRUE)
  n2 <- all.vars(e, unique=TRUE)
  setdiff(n1, c(n2, "+", "-", "*", "/", "^", "=", "("))
}

#### Var2Array
#### walk a parse tree, making variable names that appear in a vector just named

```

```

### arguments in a vector whose name is passed as an argument. For example,
### suppose x <- parse(text="A <- B * C")
### x <- Var2Array(x,c("A","B","C"),"Pm")
### would result in:
### > deparse(x)
### > Pm["A"] <- Pm["B"] * Pm["C"]

Var2Array <- function(x, vars, aname) {
  if (length(x) == 1) {
    if (as.character(x) %in% vars) {
      parse(text=paste(aname,"[\"",x,"\""],sep=""))[[1]]
    } else x
  } else {
    for (i in 1:length(x)) x[[i]] <- Var2Array(x[[i]],vars,aname)
    x
  }
}
### Some functions to process documentation comments
###
### getDocTag returns the value associated with tag "tag", presumed to
### be a single line of text.
### ode is a vector of strings.

getDocTag <- function(ode,tag)
{
  ## create search string from tag
  regex <- paste("@",tag,
                ":[[:space:]]*",sep="")
  ## Get lines with the tag
  ln <- grep(regex,ode,ignore.case=TRUE)
  if (length(ln) > 0) {
    sub(paste(regex,"(.*)",sep=""),"\\"1",ode[ln[1]],ignore.case=TRUE)
  } else paste("No",tag)
}
### getLongDoc returns longer documentation blocks that begin and end with
### @BEGIN "tag"
### ...
### @END "tag"
### Just as in other blocks, these BEGIN and END keys need to be on lines by themselves.
getLongDoc <- function(ode,tag)
{
  startline <-
  grep(paste("@BEGIN[[:space:]]*",tag,sep=""),ode,ignore.case=TRUE)
  if (length(startline) > 0) {
    startline <- startline[1] + 1
    endline <- grep(paste("@END[[:space:]]*",tag,sep=""),ode,ignore.case=TRUE)
    if (length(endline) > 0) {
      endline <- endline[1] - 1
    } else {
      stop(paste("Documentation block",tag,"does not end with @END",tag))
    }
    ode[startline:endline]
  } else paste("No",tag)
}

### makeDep takes a list of parsed statements and constructs a

```

```

### dependency structure.
### this is a list, each element of which is a list of two elements: the first
### is the lhs of a statement, and the second is a character vector
### of all the variables on the rhs
### this uses R's 'all.variables()' idea of what a variable is.
makeDep <- function(p)
{
  lapply(p,function(x)list(all.vars(x[[2]]),all.vars(x[[3]])))
}

### orderStatements(depends) returns the index vector required to order
### statements so that any variable is used only after it is defined, AND
### identify when there are circularities so that this cannot be accomplished.
###
### The algorithm is due to Ramon Garcia. Start with a set of defined terms,
### 'Defined' and a list to hold the sorted statements. Add all statements
### whose rhs is composed solely of elements of Defined to the sorted list,
### and add their lhs to 'Defined'. Now repeat this step until all statements
### have been moved to the sorted list. This has to work, because if at some
### step it is not possible to move statements to the sorted list, there must
### be a circularity in the definitions, which we have already determined
### does not exist.

### The return value is an index vector, such that if Statements is a list of
### (possibly parsed) statements, Statements[indx] is in the right order.

orderStatements <- function(dp, Defined=NULL)
{
  ## Assign each statement its own index, which will move with it.
  for (i in 1:length(dp)) dp[[i]]$Index <- i

  ## list to hold the new ordering
  ordered.dp <- vector("list",0)
  ## First identify
  ## all statements that have only constants on the rhs, and add their lhs
  ## to Defined
  ldep <- sapply(dp,function(x)length(x[[2]]))
  newsub <- dp[ldep == 0]
  dp <- dp[ldep > 0]
  ordered.dp <- c(ordered.dp,newsub)
  ## Extract the lhs's and add to Defined
  Defined <- c(Defined,sapply(newsub, function(x)x[[1]]))
  while (length(dp) > 0) {
    ## Find the statements whose rhs's are completely defined already
    AllDef <- sapply(dp,function(x)all(x[[2]] %in% Defined))
    newsub <- dp[AllDef]
    ordered.dp <- c(ordered.dp, newsub)
    dp <- dp[!AllDef]
    Defined <- c(Defined, sapply(newsub, function(x)x[[1]]))
  }
  ## extract the indices for ordering
  sapply(ordered.dp, function(x)x$Index)
}

### depcheck(lst, nm1, nm2) checks for circularities in a dependencies list
### like that used as input to orderStatements. This is part of a basic
### check for modeling errors, and is always run.

```

```

depcheck <- function(lst, nm1, nm2)
{
  if (!is.null(nm1)) {
    if (length(lst[[nm1]]) == 0) return(TRUE)
    if (nm2 %in% lst[[nm1]]) return(FALSE)
    out <- vector("logical", length(lst[[nm1]]))
    names(out) <- lst[[nm1]]
  } else {
    out <- vector("logical", length(lst))
    names(out) <- names(lst)
  }
  ## Come here on initial call
  for (f in names(out)) {
    if (is.null(nm1)) nm2 <- f
    out[f] <- depcheck(lst,f,nm2)
  }
  all(out)
}

### getDocStr(x):
###   x is a string of statements that begin "zzz =" and end in "@ xyw"
###   where "zzz" is a name, and "xyw" is a documentation string for that name
###   getDocStr(x) returns all the documentation strings in a vector whose name
###   attribute is the vector of names. That is, if a line was:
###   PDo = 27.9 @ blood diaphragm PC for oxon
###   the return value would be:
###   Docstr == c(PDo = "blood diaphragm PC for oxon")
getDocStr <- function(x)
{
  x <- x[grep("@",x)]
  if (length(x) > 0) {
    DocStr <- sub(".*@[:space:]*(.*)","\\1",x)
    VarNames <-
    sub("^([:alpha:]][[:alnum:]_]*)(\\[[[:alnum:]_]+[:space:]]*([[:space:]]*[:space:]*[:alnum:]_+)*\\)?[:space:]*[.;:=].*","\\1",x)
    names(DocStr) <- VarNames
    DocStr
  } else NA
}

### cleanDocStr(x)
###   trims off the doc string from input lines, and removes any resulting
###   trailing white space.
cleanDocStr <- function(x)
{
  sub("[[:space:]]*@\.""$","",x)
}

### x is a single string containing an ARRAY declaration. Return value is a list
### with a name attribute. Names are the (unsubscripted) variable names.
### elements are lists; first element of each list is a vector of dimensions,
### and the second element is a (possibly empty) documentation string.
doArray <- function(x) {
  x <- sub("[[:space:]]*ARRAY[:space:]*","",x)
  ## split between the declarations, at the comma. need to distinguish the comma within the dimension
}

```

with the comma as separator.

```

indim <- FALSE
for (i in 1:nchar(x)) {
  indim <- switch(substr(x, i, i),
    "["=TRUE,
    "]"=FALSE,
    indim)
  if (!indim && (substr(x,i,i) == ","))
    substr(x,i,i) <- "\n"
x <- strsplit(x, "\n")[1]
## drop blank lines
x <- x[x != ""]
## split at '@' symbol
x <- strsplit(x, "@")
y <- lapply(x, function(z) {
  ## get the name
  nm <- sub("[[:space:]]*([[:alpha:]_][[:alnum:]_]*).*","\\1",z[1])
  ## get the dimensions
  dim <- eval(parse(text=sub(".*\\"[[[:digit:]]+([[:digit:]]+)*]\\","c("\\1",z[1]))))
  ## Remove leading whitespace in doc
  z[2] <- sub("^[[[:space:]]*", "", z[2])
  ## and trailing
  z[2] <- sub("[[:space:]]$","",z[2])
  list(name=nm, dim=dim, doc=z[2])
})
structure(lapply(y, function(z) z[2:3]), names=sapply(y, function(z) z$name))
}

doKeep <- function(x) {
  ## remove the keyword and surrounding space
  x <- sub("[[:space:]]*KEEP[[[:space:]]*","",x)
  ## split between the declarations, at the comma.
  ## split at any comma (so commas should not be use in the doc strings)
  x <- strsplit(x, ",")[1]
  ## drop blank lines
  x <- x[x != ""]
  ## split at '@' symbol
  x <- strsplit(x, "@")
  y <- lapply(x, function(z) {
    ## get the name
    nm <- sub("[[:space:]]*([[:alpha:]_][[:alnum:]_]*)[[:space:]]","\\1",z[1])
    ## Remove leading whitespace in doc
    z[2] <- sub("^[[[:space:]]*", "", z[2])
    ## and trailing
    z[2] <- sub("[[:space:]]$","",z[2])
    list(name=nm, doc=z[2])
  })
  structure(sapply(y, function(z) z[2]), names=sapply(y, function(z) z$name))
}
### p is a vector of statements such as would be in CONTINUOUS or
### an EVENT.
### loop through the elements of p.
### q <- p[i]
### if q contains 'FOR (',
### 1) find the end of the expression : count '{' and '}'
### 2) increment i to point to the element after the terminating '}'

```

```

### 2) replace q with the result of expandFor on the compound
###   expression following 'FOR \\(.*)\\)'
### if lis.null(index), replace the index with values from istart to istop
### in turn in q
### return the resulting vector
expandFor <- function(p, index=NULL, istart=NA, istop=NA) {
  x <- character(0)
  i <- 1
  while (i <= length(p)) {
    q <- p[i]
    if (length(grep("^[:space:]*FOR[:space:]*\\(.*)\\)", q)) > 0) {
      ## Parse the 'FOR' arguments
      ## FOR (i in istart:istop)
      out <- parseFOR(q)
      ## find the end of the FOR block
      itop <- i <- i + 1
      nlevel <- 1
      while (nlevel > 0) {
        i <- i + 1
        r <- p[i]
        for (j in 1:nchar(r)) {
          if (substr(r, j, j) == "{") {
            nlevel <- nlevel + 1
            break
          }
          if (substr(r, j, j) == "}") {
            nlevel <- nlevel - 1
            break
          }
        }
      }
      ## expand the FOR block
      q <- expandFor(p[(itop+1):(i-1)], index=out$index, istart=out$istart,
                     istop=out$istop)
    }
    ## append the new (possibly expanded) block to the existing block, and look at the next
    ## statement
    x <- c(x,q)
    i <- i + 1
  }
  ## substitute the index
  if (!lis.null(index)) {
    dosubst(x, index, istart, istop)
  } else {
    x
  }
}
## dosubst replaces the string pointed to in index
## with the values from istart:istop, consecutively
dosubst <- function(q, index, istart, istop) {
  repl <- paste("\\<", index, "\\>", sep="")
  out <- character(0)
  for (i in istart:istop)
    out <- c(out, gsub(repl, as.character(i), q))
  out
}

```

```

## parseFOR parses a line like 'FOR (I in 3:12)' and returns
## the values index="I", istart=3, istop=12.

parseFOR <- function(x) {
  out <-
  gsub("^\[[\[:space:]\]]*\[FOR\[[\[:space:]\]]*\\\\[([\[:space:]\]]*([\[:alpha:]\_][\[:alnum:]\_]*[\[:space:]\]]*((IN)|(in))[\[:space:]\]]*(\[[\[:digit:]\]+)\[[\[:space:]\]]*([\[:digit:]\_]+)\[[\[:space:]\]]*\\\].*$","\\1 \\5 \\6",x)
  tmp <- strsplit(out," ")[1]
  list(index=tmp[1], istart=as.numeric(tmp[2]), istop=as.numeric(tmp[3]))
}

### evalSubs looks at each element in a vector of statements, and evaluates numeric
### subscripts so they are simple integers:
### e.g. Var[2+1, 3-1] -> Var[3,2]
### Then, it converts to internal variable name format, with underscores.
### For simplicity, only '+' and '-' are allowed in subscripts

evalSubs <- function(p) {
  ### where are the statements in p that have subscripts
  indx <- grep("\[[\[:space:]\]]*\[([[\[:digit:]\_+-]\]+)\[[\[:space:]\]]*\[([[\[:digit:]\_+-]\]+)\]*[\[:space:]\]]*\\\]",p)
  for (i in seq(along=indx)) {
    q <- strsplit(p[indx[i]]," ")[1]
    r <- character(0)
    ## Evaluate the subscripts
    insub <- FALSE
    k <- 1
    for (j in seq(along=q)) {
      if (insub) {
        if (q[j] == ",") {
          substop <- j - 1
          r[k] <-
            as.character(eval(parse(text=paste(q[substart:substop],
                                              collapse=""))))
          k <- k+1
          r[k] <- q[j]
          k <- k+1
          substart <- j+1
        } else {
          if (q[j] == "]") {
            substop <- j - 1
            r[k] <-
              as.character(eval(parse(text=paste(q[substart:substop],
                                                collapse=""))))
            k <- k+1
            r[k] <- q[j]
            k <- k+1
            insub <- FALSE
          }
        }
      } else {# !insub
        if (q[j] == "[") {
          insub <- TRUE
          substart <- j+1
          r[k] <- q[j]
          k <- k+1
        } else {
      }
    }
  }
}

```

```

        r[k] <- q[j]
        k <- k+1
    }
}
q <- paste(r, collapse="")
## switch from bracket-form to underscore form
# q <- gsub("[,_]", "_", gsub("\\" "", q))
## replace in the original vector
p[indx[i]] <- q
}
p
}

### modified package.skeleton that makes the directory structure without the
### "chatter" or extra files of package.skeleton()

silent.package.skeleton <- function(name="anRpackage",
                                    path=".",
                                    force = FALSE)
{
  safe.dir.create <- function(path) {
    dirTest <- function(x) !is.na(isdir <- file.info(x)$isdir) &
      isdir
    if (!dirTest(path) && !dir.create(path))
      stop(gettextf("cannot create directory '%s'", path),
           domain = NA)
  }
  curLocale <- Sys.getlocale("LC_CTYPE")
  on.exit(Sys.setlocale("LC_CTYPE", curLocale), add = TRUE)
  if (Sys.setlocale("LC_CTYPE", "C") != "C")
    warning("cannot turn off locale-specific chars via LC_CTYPE")
  if (file.exists(file.path(path, name)) && !force)
    stop(gettextf("directory '%s' already exists", name),
         domain = NA)
  safe.dir.create(file.path(path, name))
  safe.dir.create(file.path(path, name, "man"))
  safe.dir.create(file.path(path, name, "src"))
  safe.dir.create(file.path(path, name, "R"))
  safe.dir.create(file.path(path, name, "data"))
}

```